MRC Technical Summary Report # 1731

SOFTWARE FOR INTERVAL ARITHMETIC: A REASONABLY PORTABLE PACKAGE

J. M. Yohe

12

Mathematics Research Center
University of Wisconsin—Madison
610 Walnut Street
Madison, Wisconsin 53706

See form 1473

March 1977

(Received February 25, 1977)

Sponsored by
U. S. Army Research Office
P. O. Box 12211
Research Triangle Park
North Carolina 27709

UNIVERSITY OF WISCONSIN - MADISON
MATHEMATICS RESEARCH CENTER

SOFTWARE FOR INTERVAL ARITHMETIC: A REASONABLY PORTABLE PACKAGE

J. M. YOHE

Technical Summary Report # 1731
March, 1977

ABSTRACT

We discuss the design and capabilities of a package of FORTRAN subroutines for performing interval arithmetic calculations. Apart from a relatively small number of primitives and constants, the package is directly transferrable to most large scale computers, and has been successfully implemented on IBM, CDC, and Honeywell equipment in addition to the UNIVAC 1110.

This package has been designed so as to be compatible with the AUGMENT precompiler, and includes interval analogs of appropriate standard FORTRAN operations and functions, as well as operations and functions peculiar to interval arithmetic. The result is that the user who has access to AUGMENT may write programs using interval arithmetic just as though FORTRAN recognized INTERVAL as a standard data type.

AMS(MOS) Subject Classification: 94-04

Key Words: Interval Arithmetic Program Package
Portable software

Work Unit No. 8 (Computer Science)

Sponsored by the United States Army under Contract Number DAAG29-75-C-0024 and a grant from the Waterways Experiment Station, Vicksburg, Mississippi.

SOFTWARE FOR INTERVAL ARITHMETIC: A REASONABLY PORTABLE PACKAGE

J. M. YOHE

1. Introduction: One means of bounding the error in digital computation is through the use of interval, or range, arithmetic; instead of computing with approximate real numbers, one calculates with pairs of approximate real numbers -- the first member of a pair being a lower bound for the true result, and the second an upper bound. By this method, one can take into account such varied sources of error as uncertainty in input data, inaccuracies in mathematical formulae, and errors in approximation of real numbers and the operations on them. The theory of interval arithmetic is developed extensively elsewhere [5]; we shall not treat it here.

The major obstacle to the use of interval arithmetic is the unavailability of software. INTERVAL is not a standard data type in any production language that we know of; preparation of a package of subprograms to handle interval data is a nontrivial task. Since the representation of and operations on interval data are necessarily rather heavily dependent upon the architecture of the host computer, a package developed for one system can not, in general, be moved intact to a different system.

This paper describes an interval arithmetic package for use with FORTRAN. The power of the AUGMENT precompiler [2, 3] is employed to render the major part of the package independent of specific data representations, and the package is so designed that the parts which are representation-dependent are concentrated in a relatively small number of modules, most

Sponsored by the United States Army under Contract Number DAAG29-75-C-0024 and a grant from the Waterways Experiment Station, Vicksburg, Mississippi.

of which are easily adapted to new environments.

Although this package was written for the UNIVAC 1110, it has also been implemented on IBM, DEC, Honeywell, and CDC equipment. No major problems have been reported in transporting the package to these host systems.

The information given in this paper is not intended to be exhaustive. The interested reader will find detailed information on all aspects of the package in the technical manual [9].

2. Design of the package:

The viewpoint taken was that of the end user. We sought to make the package *complete, accurate, convenient to use, fail-safe,* and *transportable.*

Completeness: All appropriate ANSI Standard Fortran [1] operations and functions were implemented, along with some (such as tangent, hyperbolic sine, and hyperbolic cosine) which are not ANSI Standard but are normally implemented in the FORTRAN language anyhow. Since interval numbers can be regarded in a natural sense as belonging to an extension of the real number system, most arithmetic operations and special functions are meaningful. In addition, there are a large number of functions peculiar to interval arithmetic (such as union and intersection of intervals, midpoint, and half-length) which were also included in the package. Finally, input/output routines and conversions between intervals and standard data types (where appropriate) were implemented. A list of the functions and operations is given in Table 2.1.

Accuracy: It is well known that error is inherent in digital computations, and that most computer architectures are less than optimal from this point of view. (Recently, there has been increased interest in developing more hospitable architecture; see, for example, Lang and Shriver [4] and Ris [6].) Moreover, it is extremely difficult, if not impossible, to obtain the information required for rigorous bounding of hardware operations. Since interval arithmetic tends to be pessimistic anyhow, we felt that the calculation of bounds through straightforward application of *a priori* estimates such as Wilkinson's [7] would lead to intolerable inaccuracy. In addition, vital information concerning such phenomena as exponent range faults is generally not available in existing systems. Consequently, the package was designed to be based on a set of arithmetic primitives of the type described in [8].

The special functions pose a different problem. A straightforward application of interval arithmetic to the algorithms used to compute these functions will yield unacceptably wide intervals, due to the dependency problem [5]. We addressed this problem by employing higher precision functions, bounding the results on the basis of accuracy information provided by the software supplier. This must be regarded as being less than completely satisfactory, since available error information is often sketchy and may not be completely rigorous; however, the bounding procedure takes these disadvantages into account, and the results can be regarded as being valid with extremely high probability.

Like the arithmetic routines, input/output routines need to be written from the ground up. Conversion routines supplied with standard FORTRAN systems have no provisions for obtaining the required bounds; moreover, most of them are of unknown, if not dubious, accuracy.

Convenience: By itself, no collection of routines to perform nonstandard arithmetic is really convenient to use. Each operation must be performed by a call on one of the subprograms in the package; this means that the user must parse every expression himself and write his program in what amounts to assembly language. The best that can be done in this setting is to minimize the inconvenience. To this end, we have kept the package as internally consistent as possible. All entry points to the package bear the prefix INT; routines used by the package itself are prefixed with INT or BPA, according to their level. Thus, by avoiding variable and subprogram names beginning with these prefixes, the user may be assured of avoiding conflicts.

Calling sequences for the routines in the package are consistent and concise. No information is transmitted which is not absolutely essential to the function being performed. Where the result of a function or operation is a standard data type, the routine is implemented as a function of that type; otherwise, the routine is implemented as a subroutine. In the former case, the arguments are simply the operands; in the latter case, the arguments are the operands together with the result. (In the interest of flexibility, the endpoints of an interval are regarded as being a nonstandard data type.)

Convenience of use of any nonstandard data type is increased dramatically by the use of an appropriate precompiler. This package is specifically designed to be used with the AUGMENT precompiler, which allows the source FORTRAN code to be written as though FORTRAN recognized INTERVAL as a standard data type. In this case, just as above, the user must avoid conflicts with the package; although the source code will not contain references to the routines of the package, the output from AUGMENT, of course, will. In addition, the user must also avoid the function names and operators shown in the table, since these become reserved words in the extension of FORTRAN. In most cases, this should not be an onerous task.

Fail-safe: Errors can occur in many of the operations of the interval package, just as they can in REAL operations. It is our viewpoint that errors should not be ignored. Each subprogram in which an error can occur will call the error-handling routine, INTRAP, prior to returning control to the calling program. If no error has occurred, INTRAP simply returns control to the routine which called it. Otherwise, INTRAP takes the action specified by a table which resides in a COMMON block; the response depends on the error which has occurred, but usually includes a print-out which gives the user complete information on the error. The user may, if he chooses, alter the response by changing the table.

Transportability: Transportability and flexibility of representation are closely linked. The package is based on three data types: BPA (mnemonic for Best Possible Answer), which is the data type of the endpoints of intervals, but is otherwise undefined except in a few primitive routines; INTERVAL, which is defined to be a BPA array of length 2; and EXTENDED, which is the data type in which evaluations of special functions are performed. In the UNIVAC version of the package, the representation of BPA is the same as that of REAL, and EXTENDED is a synonym for DOUBLE PRECISION.

The AUGMENT precompiler is used to extend the representations of these nonstandard data types throughout the package. The output of the AUGMENT precompiler is a set of routines which, apart from the arithmetic primitives which are written in assembly language, conforms as closely as possible to ANSI Standard FORTRAN.

There are less than twenty program modules which depend on the representations of BPA and EXTENDED numbers; many of these will need no alteration for most applications. Adaptation of the package to other hardware is discussed more fully in Section 4.

3. Use of the INTERVAL package:

If used as a collection of subroutines, without the benefit of the AUGMENT precompiler, the INTERVAL package is, of course, used just as any package of subprograms would be used. That is, the user must decide which routines must be invoked and in what order. We prefer to regard the INTERVAL package as an extension of the capabilities of the host computer system, and the AUGMENT precompiler as an instrument for extending FORTRAN to take advantage of the additional power. Thus we will address the question of use of the package in this context; the user who by reason of preference or necessity does not use the precompiler will have no difficulty in adapting this discussion to his needs.

Type declarations for INTERVAL variables: If X, Y, and Z represent INTERVAL variables, they must be declared as such by the statement

```
INTERVAL X, Y, Z
```

INTERVAL variables may be dimensioned; the only restriction is that if the FORTRAN compiler limits the number of dimensions of an array, that limit must be decreased by 1 for INTERVAL variables. The reason for this is that AUGMENT will declare INTERVAL variables as arrays.

Assignment of values to INTERVAL variables: Most real numbers can not be represented exactly in the computer. The error inherent in a statement such as

```
X = .1
```

may not be immediately obvious. If X is an INTERVAL variable, the above statement will assign a value to X, but that value will not, in general, be an interval containing the real number .1. In order to set X to an interval which does contain .1, one may write

```
X = '(.1, .1)$', or    X = 9H(.1, .1)$    if the host compiler
```

does not accept quoted Hollerith literals. If the host compiler generates a sentinel for a Hollerith literal, and if the UNPACK primitive recognizes that sentinel, the terminal $ may be omitted. Any string that is legal input for the formatted read (see discussion below and Appendix 1) is also acceptable to the routine which performs this conversion. Thus, on the UNIVAC 1110, the statement

```
X = '.1'
```

would also have the desired effect.

Reading INTERVAL variables: Two options are available in this package: a free format read and a formatted read.

The free format read will obtain the next data field from the input stream on the specified unit, convert it, and store the result in the specified INTERVAL variable. The calling sequence is

```
CALL INTRDF(UNIT, X)
```

The basic package will recognize units 5 (standard input) and 0 (reread), but the user may add other units or change unit designations as desired; this is discussed in the technical documentation. A data field may be any legal representation of an interval variable (see Appendix 1); however, for simplicity, one may be assured that the format (*number, number*), where *number* is any legal FORTRAN string representing an integer, fixed point number, or floating point number, is always valid. Embedded blanks between matching parentheses are *always ignored*. Fields may be separated by blanks (as many as desired), although if intervals are enclosed in parentheses as indicated above, blanks are unnecessary. Fields may be continued across card boundaries. The input stream remains uninterrupted so long as all reading is done by INTRDF and the unit number does not change. Once the input stream has been interrupted, INTRDF begins a new input stream with a new record.

The formatted read, as its name implies, reads interval data according to a specified format. This routine reads a vector of values (which may be of length 1). The calling sequence is

```
        CALL INTRD(UNIT, FMT, A, N)
```

Unit is as in the free format read; A is the first location of the vector into which the data is to be read; and N is the length of the vector. FMT is an array of length 3; FMT(1) is the number of data items per record, FMT(2) is the number of characters to be ignored before each data field, and FMT(3) is the width of each data field. Note that these values are constant for each call to INTRD. A data field may be any legal representation of an interval variable; parentheses are optional, and embedded blanks are permitted. No other information is permitted within a data field.

Computing with INTERVAL variables: Expressions involving INTERVAL variables are written in standard FORTRAN syntax, just as though INTERVAL were a standard FORTRAN data type. A list of the operations and functions available in this package may be found in Appendix 2.

Mixed mode expressions are permitted, but their use is discouraged due to the high probability of introducing hidden error. For example, the expression

```
        Y = 0.1 * X
```

where X and Y are INTERVAL variables, will not yield a correct value of Y; 0.1 will first be converted to REAL by the compiler, and AUGMENT will then cause that REAL number to be converted to a degenerate interval not containing .1. Multiplication will then occur using this erroneous interval.

Other operators and functions peculiar to interval arithmetic are implemented; examples include the intersection of two intervals, the union of two intervals, derivation of the midpoint and half-length, etc. These are listed in Appendix 2. Relational operators are also implemented, but they take on different meanings in the context of interval arithmetic; see Appendix 2 for details.

Writing INTERVAL variables: The write routine will convert a vector (possibly of length 1) of INTERVAL variables to external format and write it on the specified output unit according to the given format. The external representation of each interval is guaranteed to contain the interval, and is the smallest interval representable in the given format which does so. The calling sequence is

```
CALL INTWR(UNIT, FMT, A, N)
```

The basic package will recognize units 6(standard printer) and 1 (standard punch), but again the user may change designations and/or add units at will. If an illegal output unit is specified, INTWR will use the standard printer instead.

FMT is now an integer array of length 4. The first three values are the same as for INTRD (except that ignored characters in the output record are filled with blanks); FMT(4) is a carriage control character for use where appropriate. This character must be either '0' or ' ', denoting double spacing or single spacing, respectively. *The width of each data field specified by the format must be at least great enough to permit the package to convert one significant digit;* in the 1110 version, this is 15 characters, assuming a 2-digit exponent. Add 2 characters for each additional exponent digit in the external format. If an illegal format is specified, the routine will default to a standard format.

A and N are as in the formatted read.

Errors: The package is designed to detect all errors as they occur. The user may elect any of the available responses for any possible error (See Appendix 3); however, the default response is to print an error message and halt the computation except in those cases where viable alternatives exist. Those cases are few indeed; they comprise arithmetic underflows (where the offending value is set either to zero or to the properly-signed number of smallest magnitude, as appropriate) and errors occurring on output (where the write routine uses standard modes of output rather than electing to scrub the computation and lose the output altogether). In the former case, the computation proceeds without notice to the user; in the latter case, a message is printed after the output is complete. The method of changing the default responses to errors is discussed in the technical documentation.

Producing an object program: Unless a sophisticated job control language allows for an automatic (from the user's point of view) invocation of the AUGMENT precompiler and the FORTRAN compiler, the generation of an object program is a two-step procedure:

1. Use AUGMENT to translate the source program into a FORTRAN program compatible with the compiler. This can be accomplished with a run stream of the following type:

```
invoke AUGMENT
description decks for BPA and INTERVAL (supplied with the package)
*BEGIN
source program
*END
```

AUGMENT will write the translated program on Unit 20.

2. Compile the output of AUGMENT using the standard FORTRAN compiler and execute the resulting program in the usual manner. The user must insure that the BLOCK DATA modules are included when the program is processed by the linkage editor.

4. Adaptation of the package:

Adaptation of the package to other hardware is not difficult provided one has access to the AUGMENT precompiler. The necessary steps are:

1. Decide on data representations for the interval endpoints and for EXTENDED precision numbers.
2. Code or revise primitives, as necessary.
3. Process the package through the AUGMENT precompiler and compile the resulting FORTRAN code.
4. Check the package.
5. Tune and recheck the package.

We discuss each of these steps in greater detail.

Data representations: Normally, the representation for interval endpoints will be the same as REAL and EXTENDED will be the same as double precision. These choices will simplify the adaptation of the package; however, for special purposes such as higher precision interval arithmetic, other choices may be made. There are several implicit assumptions which will, to a certain extent, govern the choices of representations:

a. The portion of the package which performs endpoint evaluations (known as type BPA) will contain explicit routines to perform all operations. As designed, it is assumed that conversion from BPA to REAL is exact, although conversion in the other direction need not be. This is done to facilitate adaptation to two's complement hardware, where the negative of a real number is not necessarily representable; we assume that the negative of every BPA number is representable.

b. It is assumed that EXTENDED is bound to a higher precision than is BPA. Moreover, we assume that every BPA number and every FORTRAN integer can be represented exactly in EXTENDED format. For the evaluation of special functions, we assume that a complete supporting package exists for type EXTENDED, and that bounds on the accuracy of these routines are available.

Primitives: There are nineteen primitives which depend on the representation of BPA and EXTENDED numbers in the host system. Two of these are BLOCK DATA modules, which contain various representation dependent constants; eight are written in FORTRAN and depend only on BPA format being the same as REAL and EXTENDED being the same as DOUBLE PRECISION; three depend on both data representations and the (nonstandard) FLD function; one contains FORMAT statements which may be representation dependent; and five are arithmetic primitives which must necessarily be recoded for any change in data representation. The arithmetic primitives are, in fact, written in assembly language.

In addition, INTRD and INTRDF, while not technically primitives, contain nonstandard READ statements which recognize the END OF FILE condition. If the host compiler does not recognize this form of READ statement, those statements will need to be modified.

Complete documentation of these primitives is given in the technical manual. It does not seem appropriate to go into greater detail here.

AUGMENT processing: The use of the AUGMENT precompiler preserves both naturality of expression and flexibility. Most of the INTERVAL package is written in terms of the nonstandard types BPA, EXTENDED, and INTERVAL. The binding to specific data representations is accomplished through the primitives, and these bindings are extended through the remainder of the package by the use of AUGMENT. Every effort has been made to write the package so that the output of the AUGMENT precompiler will be ANSI Standard FORTRAN. There is no requirement that AUGMENT be available on the target computer; the preprocessing can just as well be done on any computer, with the resulting

FORTRAN code being brought to the target system for compilation.

Checking the package: A collection of test programs is provided with the INTERVAL package. Successful execution of these programs is reasonably good assurance that the primitives have been implemented properly.

Tuning the package: The price paid for the degree of flexibility present in the source code for this package is quite likely to be decreased efficiency in the object code. For example, since the format of BPA numbers is arbitrary, conversion from REAL to BPA will generate a call on a subprogram which is responsible for performing this task (this subprogram is, of course, a primitive). If BPA numbers are the same as REAL, this will result in unnecessary overhead; an in-line replacement operation would perform the same task at considerably less cost. AUGMENT can not be instructed to make this modification; thus, for greatest efficiency, it will be necessary to examine the output of AUGMENT and replace calls of this type by in-line replacement statements. There are, of course, many other possibilities, depending on representation; for example, if the hardware has double precision capability, one could change calls on the interval replacement subroutine to in-line replacement statements using the double precision hardware.

A certain amount of care must be exercised in tuning the package. For example, the routines which evaluate BPA relational operators call on the BPA subtract routine. This should not be altered unless the hardware subtract always produces a result of the same sign as the true result, even in cases of underflow and overflow. If the hardware sets an underflow to zero, or gives garbage when overflow occurs, then the hardware subtract must not be used.

Needless to say, the package must be rechecked whenever any changes are made.

5. *Conclusion:*

In this paper, we have sketched the design and use of a package for performing calculations in interval arithmetic. The package is both flexible and transportable; adaptation of the package to other systems can be accomplished by rewriting a maximum of nineteen primitive modules, most of which are easily adapted to a new host system. Further details of the package are provided in the technical documentation.

REFERENCES

1. *ANSI Standard FORTRAN*, American National Standards Institute, New York, 1966.

2. Crary, F. D. The AUGMENT precompiler I. User information. The University of Wisconsin - Madison, Mathematics Research Center, Technical Summary Report # 1469, December, 1974.

3. ____________. The AUGMENT precompiler II. Technical documentation. The University of Wisconsin - Madison, Mathematics Research Center, Technical Summary Report # 1470,

4. Lang, Allan L. and Shriver, Bruce D. The design of a polymorphic arithmetic unit. *Third IEEE - TCCA Symposium on Computer Arithmetic*, November, 1975, 48 - 55.

5. Moore, Ramon E. *Interval Analysis*. Prentice - Hall, Inc., Englewood Cliffs, N. J., 1966

6. Ris, Frederic N. A unified decimal floating-point architecture for the support of high-level languages (extended abstract). *SIGNUM Newsletter 11*, 3 (October, 1976), 18 - 22.

7. Wilkinson, J. H. Rounding errors in algebraic processes. *Notes on Applied Science No. 32*, Her Majesty's Stationery Office, London, 1963.

8. Yohe, J. M. Roundings in floating-point arithmetic. *IEEE Trans. Computers C-22* (1973), 577 - 586.

9. __________. The INTERVAL Arithmetic package. The University of Wisconsin - Madison, Mathematics Research Center, Technical Summary Report (forthcoming).

APPENDIX 1

STANDARD FORTRAN NUMBER AND INTERVAL NUMBER REPRESENTATIONS

```
DIGIT          ::=  0|1|2|3|4|5|6|7|8|9

SIGN           ::=  +|-

INTEGER        ::=  NULL|<SIGN>|<INTEGER><DIGIT>

RADIX          ::=  .

FIXEDPOINT     ::=  <INTEGER><RADIX>|<FIXEDPOINT><DIGIT>

EXPSEP         ::=  E|D

EXPONENT       ::=  <SIGN>|<EXPSEP>|<EXPSEP><SIGN>|<EXPONENT><DIGIT>

NUMBER         ::=  <INTEGER>|<FIXEDPOINT>|<INTEGER><EXPONENT>|

                    <FIXEDPOINT><EXPONENT>

ENDPTSEP       ::=  :

COMMA          ::=  ,

INTERVAL       ::=  <NUMBER>|(<NUMBER>)|<NUMBER><ENDPTSEP><NUMBER>|

                    (<NUMBER><ENDPTSEP><NUMBER>)|(<NUMBER><COMMA><NUMBER>)
```

INTERVAL INPUT RULES

FORMATTED INPUT:

One and only one <INTERVAL> shall appear in any one field.

Embedded blanks are permitted; they will be *ignored*.

FREE FORMAT INPUT:

Leading blanks are *always ignored*.

Blanks within matching pairs of parentheses are *always ignored*.

Commas within matching pairs of parentheses are regarded as endpoint separators.

A field consists of exactly one <INTERVAL>.

A field is terminated by

1. A visible blank;

2. Any of the characters '$', '#', '=';

3. A comma occurring outside of a matching pair of parentheses;

4. Any nonblank character following a matching right parenthesis (If such character is not '$', '#', '=', or ',', it will be regarded as the first character of the next field);

5. A left parenthesis or colon occurring outside of a matching pair of parentheses. (Such character will be regarded as the first character of the next field).

If a left parenthesis is encountered, the scan proceeds to the matching right parenthesis *regardless of what characters are encountered,* except that '$', '#', and '=' *always* terminate the field.

ALL INPUT:

A null field is taken to represent the interval (0, 0).

A field containing <NUMBER> or (<NUMBER>) is taken to represent a degenerate interval; this number is converted and rounded down for the left endpoint, and up for the right endpoint.

If a field contains two <NUMBER>s, the first will be converted and rounded down for the left endpoint, and the second will be converted and rounded up for the right endpoint.

APPENDIX 2

OPERATION	DEFINITION/EXPLANATION	RESULT TYPE	ROUTINE INVOCATION VIA AUGMENT	ROUTINE INVOCATION DIRECT	ROUTINE TYPE
ARITHMETIC					
Add	**Sum of two intervals**	**X**	**XA + XB**	**INTADD(XA, XB, XR)**	**S**
Subtract	**Difference of two intervals**	**X**	**XA - XB**	**INTSUB(XA, XB, XR)**	**S**
Multiply	**Product of two intervals**	**X**	**XA * XB**	**INTMUL(XA, XB, XR)**	**S**
Divide	**Quotient of two intervals**	**X**	**XA / XB**	**INTDIV(XA, XB, XR)**	**S**
EXPONENTIATION					
to BPA	**Raise interval to BPA power**	**X**	**XA ** BB**	**INTXXB(XA, BB, XR)**	**S**
to EXTENDED	**Raise interval to EXTENDED power**	**X**	**XA ** EB**	**INTXXE(XA, EB, XR)**	**S**
to INTEGER	**Raise interval to INTEGER power**	**X**	**XA ** IB**	**INTXXI(XA, IB, XR)**	**S**
to INTERVAL	**Raise interval to INTERVAL power**	**X**	**XA ** XB**	**INTXXX(XA, XB, XR)**	**S**
MATHEMATICAL					
Absolute value	$\{\vert x\vert : x \in XA\}$	**X**	***ABS*(XA)**	**INTABS(XA, XR)**	**S**
Arc cosine	**Arc cosine of interval XA**	**X**	***ACOS*(XA)**	**INTACS(XA, XR)**	**S**
Arc sine	**Arc sine of interval XA**	**X**	***ASIN*(XA)**	**INTASN(XA, XR)**	**S**
Arc tan (2 args)	**Arc tangent of XA / XB**	**X**	***ATAN2*(XA, XB)**	**INTAT2(XA, XB, XR)**	**S**
Arc tangent	**Arc tangent of interval XA**	**X**	***ATAN*(XA)**	**INTATN(XA, XB)**	**S**
Cube root	**Cube root of interval XA**	**X**	***CBRT*(XA)**	**INTCBT(XA, XR)**	**S**
Cosine	Cosine of interval XA	X	*COS*(XA)	INTCOS(XA, XR)	S
Hyperbolic cosine	Hyperbolic cosine of interval XA	X	*COSH*(XA)	INTCSH(XA, XR)	S
Exponential	e ↑ XA	X	*EXP*(XA)	INTEXP(XA, XR)	S
Integer	Smallest interval with integer endpoints containing the interval XA	X	*INT*(XA)	INTINT(XA, XR)	S
Natural logarithm	Log to the base e of interval XA	X	*LN*(XA) or *LOG*(XA)	INTLN (XA, XR)	S
Common logarithm	Log to the base 10 of interval XA	X	*LOG10*(XA)	INTLOG(XA, XR)	S
Sine	Sine of interval XA	X	*SIN*(XA)	INTSIN(XA, XR)	S
Hyperbolic sine	Hyperbolic sine of interval XA	X	*SINH*(XA)	INTSNH(XA, XR)	S
Square root	Square root of interval XA	X	*SQRT*(XA)	INTSQT(XA, XR)	S
Tangent	Tangent of interval XA	X	*TAN*(XA)	INTTAN(XA, XR)	S
Hyperbolic tangent	Hyperbolic tangent of interval XA	X	*TANH*(XA)	INTTNH(XA, XR)	S
FIELD					
Infimum	Left endpoint of interval XA	B	*INF*(XA)	INTINL(BA, XR) **(insertion)**	**S**
				INTINF(XA) **(extraction)**	**B**
Supremum	Right endpoint of interval XA	B	*SUP*(XA)	INTSPL(BA, XR) **(insertion)**	S
				INTSUP(XA) **(extraction)**	**B**
SPECIAL INTERVAL FUNCTIONS					
Compose	Form interval from two BPA endpoints	X	*COMPOS*(BA, BB)	INTCPS(BA, BB, XR)	S
Distance	Max($\vert$Inf(XA)-Inf(XB)$\vert$, $\vert$Sup(XA)-Sup(XB)$\vert$)	B	*DIST*(XA, XB)	INTDST(XA, XB, BR)	S
Half length	(Sup(XA)-Inf(XA))/2, rounded up	B	*HLGTH*(XA)	INTHLB(XA, BR)	S
Length	Sup(XA)-Inf(XA), rounded up	B	*LGTH*(XA)	INTLGB(XA, BR)	S

OPERATION	DEFINITION/EXPLANATION	RESULT TYPE	ROUTINE INVOCATION VIA AUGMENT	DIRECT	ROUTINE TYPE
SPECIAL INTERVAL FUNCTIONS(continued)					
Magnitude	Sup(Abs(XA))	B	*MAG*(XA)	INTMAG(XA, BR)	S
Midpoint	(Sup(XA)+Inf(XA))/2, rounded nearest	B	*MDPT*(XA)	INTMDB(XA, BR)	S
Mignitude	Inf(Abs(XA))	B	*MIG*(XA)	INTMIG(XA, BR)	S
Pivot	$\sqrt{\text{Mag(XA)}\times\text{Mig(XA)}}$, rounded down	B	*PIVL*(XA)	INTPVL(XA, BR)	S
	same, rounded as specified by IB	B	-	INTPVO(XA, IB, BR)	S
	same, rounded up	B	*PIVU*(XA)	INTPVU(XA, BR)	S
Intersection	Set-theoretic intersection of XA and XB	X	XA.*INTSCT*.XB	INTSCT(XA, XB, XR)	S
Sign	+1 if Inf(XA) > 0; -1 if Sup(XA) < 0; 0 if 0 ε XA	I	*SGN*(XA)	INTSGN(XA)	I
Size	(Abs(Inf(XA)) + Abs(Sup(XA)))/2	B	*SIZE*(XA)	INTSIX(XA, BR)	S
Union	Smallest interval containing both XA, XB	X	XA.*UNION*.XB	INTUNN(XA, XB, XR)	S
CONVERSION					
PH → X	Packed Hollerith to Interval	X	*CTX*(string)	INTASG(HA, XR)	S
E → X	Extended to Interval, bounded at IBth dig.	X	-	INTBND(EA, IB, XR)	S
B → X	BPA to Interval	X	*CTX*(BA)	INTCBX(BA, XR)	S
E → X	Extended to Interval	X	*CTX*(EA)	INTCEX(EA, XR)	S
UH → X	Unpacked Hollerith to Interval (width IB)	X	-	INTCHX(HA, IB, XR)	S
I → X	Integer to Interval	X	*CTX*(IA)	INTCIX(IA, XR)	S
R → X	Real to Interval	X	*CTX*(RA)	INTCRX(RA, XR)	S
X → B	Interval to BPA	B	*CTB*(XA)	INTCXB(XA, BR)	S
X → E	Interval to Extended	E	*CTE*(XA)	INTCXE(XA, ER)	D
X → UH	Interval to unpacked Hollerith (width IB)	UH	-	INTCXH(XA, HR, IB)	S
X → I	Interval to Integer	I	*CTI*(XA)	INTCXI(XA, IR)	I
X → R	Interval to Real	R	*CTR*(XA)	INTCXR(XA, RR)	R
SERVICE					
Store	Replacement operator	X	XR = XA	INTSTR(XA, XR)	S
Negate	Unary minus	X	-XA	INTNEG(XA, XR)	S
LOGICAL AND RELATIONAL					
Bad interval	Inf(XA) > Sup(XA)	L	*BAD*(XA)	INTBAD(XA)	L
Element of	BA ε XB	L	BA .*E*. XB	INTELE(BA, XB)	L
Good interval	Inf(XA) ≤ Sup(XA)	L	*OK*(XA)	INTOK (XA)	L
Subset of	XA contained in XB	L	XA.*SUBSET*.XB	INTSBS(XA, XB)	L
Set-equal	Inf(XA) = Inf(XB) and Sup(XA) = Sup(XB)	L	XA .*SEQ*. XB	INTSEQ(XA, XB)	L
Set-greater-equal	Sup(XA) ≥ Sup(XB)	L	XA .*SGE*. XB	INTSGE(XA, XB)	L
Set-greater	Sup(XA) > Sup(XB)	L	XA .*SGT*. XB	INTSGT(XA, XB)	L
Set-less-equal	Inf(XA) ≤ Inf(XB)	L	XA .*SLE*. XB	INTSLE(XA, XB)	L
Set-less	Inf(XA) < Inf(XB)	L	XA .*SLT*. XB	INTSLT(XA, XB)	L
Set-not-equal	Inf(XA) ≠ Inf(XB) or Sup(XA) ≠ Sup(XB)	L	XA .*SNE*. XB	INTSNE(XA, XB)	L

OPERATION	DEFINITION/EXPLANATION	RESULT TYPE	ROUTINE INVOCATION VIA AUGMENT	ROUTINE INVOCATION DIRECT	ROUTINE TYPE
LOGICAL AND RELATIONAL(continued)					
Superset	XA contains XB	L	XA.*SPRSET*.XB	INTSPS(XA, XB)	L
Value-equal	Inf(XA) = Sup(XA) = Inf(XB) = Sup(XB)	L	XA .*VEQ*. XB	INTVEQ(XA, XB)	L
Value-greater-eq.	Inf(XA) $\geq$ Sup(XB)	L	XA .*VGE*. XB	INTVGE(XA, XB)	L
Value-greater	Inf(XA) > Sup(XB)	L	XA .*VGT*. XB	INTVGT(XA, XB)	L
Value-less-equal	Sup(XA) $\leq$ Inf(XB)	L	XA .*VLE*. XB	INTVLE(XA, XB)	L
Value-less	Sup(XA) < Inf(XB)	L	XA .*VLT*. XB	INTVLT(XA, XB)	L
Value-not-equal	XA does not intersect XB	L	XA .*VNE*. XB	INTVNE(XA, XB)	L
INPUT/OUTPUT					
Read	Read interval vector, formatted	X	-	INTRD (UNIT, FMT, XR, IN)	S
Read, free format	Read next interval from data stream	X	-	INTRDF(UNIT, XR)	S
Write	Write interval vector, formatted	-	-	INTWR (UNIT, FMT, XA, IN)	S
MISCELLANEOUS					
Reduce	Reduce argument of trig function to principal range	X	-	INTRED(XA, IB, XR)	S
Unpack	Unpack packed Hollerith	H	-	INTUPK(HA, HR, IB, IC)	S
ERROR HANDLING					
Trap	Detect errors in interval package (arguments in COMMON)	-	-	INTRAP	S

NOTES ON TABLE:

DATA TYPES: B = BPA; E = EXTENDED; I = INTEGER; L = LOGICAL; R = REAL; X = INTERVAL; H = HOLLERITH; UH = UNPACKED HOLLERITH; PH = PACKED HOLLERITH; D = DOUBLE PRECISION

ROUTINE TYPES: S = SUBROUTINE; any other letter denotes a function of the indicated type (see above).

VARIABLE NAMES: The first letter indicates the type of the variable. The second letter is R for RESULT, A or B for argument; other letters may be used for special meanings.

APPENDIX 3

FAULT NUMBER	MEANING				ACTION CODE
0	No fault				0
1	Left endpoint -	no fault	Right endpoint -	overflow	4*
2		no fault		infinity	3
3		no fault		underflow	0
4		overflow		no fault	4*
5		overflow		overflow	4*
6		overflow		infinity	3
7		overflow		underflow	4*
8		infinity		no fault	3
9		infinity		overflow	3
10		infinity		infinity	3
11		infinity		underflow	3
12		underflow		no fault	0
13		underflow		overflow	4*
14		underflow		infinity	3
15		underflow		underflow	0
16	Division by zero				3
17	Zero raised to the zero power				1
18	Square root of a negative number				3
19	Logarithm of a nonpositive number				3
20	Underflow during computation of a BPA result				0
21	Overflow during computation of a BPA result				3
22	Intersection of disjoint intervals				3
23	Arc cosine or arc sine argument out of range				3
24	Inverted interval				4
25	Illegal input character				4
26	Illegal input format specification				4
27	Illegal output format specification				1
28	Input string too long				4
29	Illegal or unspecified input unit				4
30	End of file on input unit				1
31	Illegal or unspecified output unit				1
32	Conversion array overflow during base conversion				4†
33	Unrecognized error				4

* Denotes that the fault is logically impossible

† This action should not be changed, since any other action could result in a recursive call on INTRAP from INTCXH.

In the event that a fault occurs, the corresponding action code governs the response of the INTRAP routine. The action codes, and their responses, are:

0	Return to the calling program without taking any action
1	Print error message and return to the calling program
2	Print error message, trace call sequence, and return
3	Print error message, trace call sequence, step error counter in Executive program, and return
4	Print error message, trace call sequence, and halt computation

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 14 MRC-TSR-1731 9	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 SOFTWARE FOR INTERVAL ARITHMETIC: A REASONABLY PORTABLE PACKAGE.	Technical	5. TYPE OF REPORT & PERIOD COVERED Summary Report, - no specific reporting period
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 J. M. Yohe		8. CONTRACT OR GRANT NUMBER(s) 15 DAAG29-75-C-0024
9. PERFORMING ORGANIZATION NAME AND ADDRESS Mathematics Research Center, University of Wisconsin 610 Walnut Street Madison, Wisconsin 53706		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 8 (Computer Science)
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office P.O. Box 12211 Research Triangle Park, North Carolina 27709		12. REPORT DATE 11 Mar[redacted]77
		13. NUMBER OF PAGES 22
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) 12 25p.		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Interval Arithmetic Program Package

Portable Software

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We discuss the design and capabilities of a package of FORTRAN subroutines for performing interval arithmetic calculations. Apart from a relatively small number of primitives and constants, the package is directly transferrable to most large scale computers, and has been successfully implemented on IBM, CDC, and Honeywell equipment in addition to the UNIVAC 1110.

This package has been designed so as to be compatible with the AUGMENT precompiler, and includes interval analogs of appropriate standard FORTRAN operations and functions, as well as operations and functions peculiar to interval arithmetic. The result is that the user who has access to AUGMENT may write programs using interval arithmetic just as though FORTRAN recognized INTERVAL as a standard data type.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

221 200